



AP-485

**APPLICATION
NOTE**

**Intel Processor
Identification and the
CPUID Instruction**

March, 2000

Order Number : 241618-014

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel's Intel Architecture processors (e.g., Pentium® processor, Pentium processor with MMX™ technology, Pentium Pro processor, Pentium II processor, Pentium II Xeon™ processor, Pentium III processor, Pentium III Xeon™ processor and Intel Celeron™ processor) may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect IL 60056-7641

or call 1-800-879-4683
or visit Intel's website at <http://www.intel.com/>

Copyright © Intel Corporation 2000.

* Third-party brands and names are the property of their respective owners.



CONTENTS

	PAGE
1. 0. INTRODUCTION	6
1.1. Update Support.....	6
2. 0. DETECTING THE CPUID INSTRUCTION.....	6
3. 0. OUTPUT OF THE CPUID INSTRUCTION.....	7
3.1. Vendor ID String.....	7
3.2. Processor Signature	9
3.3. Feature Flags.....	12
3.4. Cache Size and Format Information.....	12
3.5. SYSENTER/SYSEXIT – SEP Features Bit	14
3.6. Pentium® Pro Processor Output Example.....	14
3.7. Pentium® II Processor, model 3 Output Example	16
4. 0. PROCESSOR SERIAL NUMBER.....	16
4.1. Presence of Processor Serial Number.....	17
4.2. Forming the 96-bit Processor Serial Number	17
5. 0. BRAND ID.....	17
6. 0. USAGE GUIDELINES	18
7. 0. PROPER IDENTIFICATION SEQUENCE.....	18
8. 0. USAGE PROGRAM EXAMPLES.....	20

REVISION HISTORY

Revision	Revision History	Date
-001	Original Issue.	05/93
-002	Modified Table 2, Intel486™ and Pentium® Processor Signatures.	10/93
-003	Updated to accommodate new processor versions. Program examples modified for ease of use, section added discussing BIOS recognition for OverDrive® processors and feature flag information updated.	09/94
-004	Updated with Pentium Pro and OverDrive processors information. Modified Tables 1, 3 and 5. Inserted Tables 6, 7 and 8. Inserted Sections 3.4. and 3.5.	12/95
-005	Added Figures 1 and 3. Added Footnotes 1 and 2. Modified Figure 2. Added Assembly code example in Section 4. Modified Tables 3, 5 and 7. Added two bullets in Section 5.0. Modified cpuid3b.ASM and cpuid3b.C programs to determine if processor features MMX™ technology. Modified Figure 6.0.	11/96
-006	Modified Table 3. Added reserved for future member of P6 family of processors entry. Modified table header to reflect Pentium II processor family. Modified Table 5. Added SEP bit definition. Added Section 3.5. Added Section 3.7 and Table 9. Corrected references of P6 family to reflect correct usage. Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code sections to check for SEP feature bit and to check for, and identify, the Pentium II processor. Added additional disclaimer related to designers and errata.	3/97
- 007	Modified Table 2. Added Pentium II processor, model 5 entry. Modified existing Pentium II processor entry to read "Pentium II processor, model 3". Modified Table 5. Added additional feature bits, PAT and FXSR. Modified Table 7. Added entries 44h and 45h. Removed the note "Do not assume a value of 1 in a feature flag indicates that a given feature is present. For future feature flags, a value of 1 may indicate that the specific feature is not present" in section 4.0. Modified cpuid3b.asm and cpuid3.c example code section to check for, and identify, the Pentium II processor, model 5. Modified existing Pentium II processor code to print Pentium II processor, model 3.	1/98
- 008	Added note to identify Intel Celeron™ processor, model 5 in section 3.2. Modified Table 2. Added Intel Celeron processor & Pentium® OverDrive® processor with MMX™ technology entry. Modified Table 5. Added additional feature bit, PSE-36. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Intel Celeron processor.	4/98
-009	Added note to identify Pentium II Xeon™ processor in section 3.2. Modified Table 2. Added Pentium II Xeon processor entry. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Pentium II Xeon processor.	6/98
-010	No Changes	

-011	Modified Table 2. Added Intel Celeron processor, model 6 entry. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Intel Celeron processor, model 6.	12/98
-012	Modified Figure 1 to add the reserved information for the Intel386 processors. Modified Figure 2. Added the Processor serial number information returned when the CUID instruction is executed with EAX=3. Modified Table 1. Added the Processor serial number parameter. Modified Table 2. Added the Pentium III processor and Pentium III Xeon processor. Added Section 4 "Processor serial number". Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code to check for and identify the Pentium III processor and the Pentium III Xeon processor.	12/98
-013	Modified Figure 2. Added the Brand ID information returned when the CUID instruction is executed with EAX=1. Added section 5 "Brand ID". Added Table 10 that shows the defined Brand ID values. Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code to check for and identify the Pentium III processor, model 8 and the Pentium III Xeon processor, model 8.	10/99
-014	Modified Table 3. Added Intel Celeron processor, model 8	03/00



1.0. INTRODUCTION

As the Intel Architecture evolves with the addition of new generations and models of processors (8086, 8088, Intel286, Intel386™, Intel486™, Pentium® processors, Pentium OverDrive® processors, Pentium processors with MMX™ technology, Pentium OverDrive processors with MMX technology, Pentium Pro processors, Pentium II processors, Pentium II Xeon™ processors, Pentium II Overdrive® processors, Intel Celeron™ processors, Pentium III processors, and Pentium III Xeon™ processors), it is essential that Intel provide an increasingly sophisticated means with which software can identify the features available on each processor. This identification mechanism has evolved in conjunction with the Intel Architecture as follows:

1. Originally, Intel published code sequences that could detect minor implementation or architectural differences to identify processor generations.
2. Later, with the advent of the Intel386 processor, Intel implemented processor signature identification that provided the processor family, model, and stepping numbers to software, but only upon reset.
3. As the Intel Architecture evolved, Intel extended the processor signature identification into the CUID instruction. The CUID instruction not only provides the processor signature, but also provides information about the features supported by and implemented on the Intel processor.

The evolution of processor identification was necessary because, as the Intel Architecture proliferates, the computing market must be able to tune processor functionality across processor generations and models that have differing sets of features. Anticipating that this trend will continue with future processor generations, the Intel Architecture implementation of the CUID instruction is extensible.

The Pentium III processors and Pentium III Xeon processors extend the concept of processor identification with the addition of processor serial number. Processor serial number is a 96-bit number accessible through the CUID instruction. Processor serial number can be used by applications to identify a processor, and by extensions, its system.

Beginning with the Pentium III processors, model 8, the Pentium III Xeon processors, model 8, and the Intel Celeron processor, model 8 the concept of processor identification is further extended with the addition of Brand ID. Brand ID is an 8-bit number accessible through the CUID instruction. Brand ID is used by

applications to identify the Intel brand name of the processor.

This application note explains how to use the CUID instruction in software applications, BIOS implementations, and various processor tools. By taking advantage of the CUID instruction, software developers can create software applications and tools that can execute compatibly across the widest range of Intel processor generations and models, past, present, and future.

1.1. Update Support

You can obtain new Intel processor signature and feature bits information from the developer's manual, programmer's reference manual or appropriate documentation for a processor. In addition, you can receive updated versions of the programming examples included in this application note; contact your Intel representative for more information, or visit Intel's website at <http://developer.intel.com/>.

2.0. DETECTING THE CUID INSTRUCTION

Starting with the Intel486 family and subsequent Intel processors, Intel provides a straightforward method for determining whether the processor's internal architecture is able to execute the CUID instruction. This method uses the ID flag in bit 21 of the EFLAGS register. If software can change the value of this flag, the CUID instruction is executable¹ (see Figure 1).

The POPF, POPFD, PUSHF, and PUSHFD instructions are used to access the Flags in Eflags register. The program examples at the end of this application note show how you use the PUSHFD instruction to read and the POPFD instruction to change the value of the ID flag.

Footnotes

¹ Only in some Intel486™ and succeeding processors. Bit 21 in the Intel386™ processor's Eflag register cannot be changed by software, and the Intel386 processor cannot execute the CUID instruction. Execution of CUID on a processor that does not support this instruction will result in an invalid opcode exception.

3.0. OUTPUT OF THE CPUID INSTRUCTION

Figure 2 summarizes the outputs of the CPUID instruction. The function of the CPUID instruction is fully dependent upon the contents of the EAX register. This means, by placing different values in the EAX register and then executing CPUID, the CPUID instruction will perform a specific function dependent upon whatever value is resident in the EAX register (see Table 1). In order to determine the highest acceptable value for the EAX register input and CPUID operation, the program should set the EAX register parameter value to “0” and then execute the CPUID instruction as follows

```
MOV EAX, 00H
CPUID
```

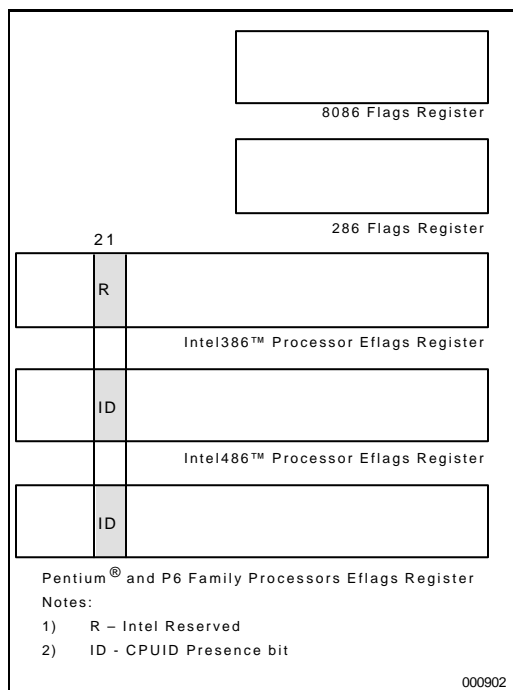


Figure 1. Flag Register Evolution

After the execution of the CPUID instruction, a return value will be present in the EAX register. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX “returned” value. On current and future IA-32 processors, bit 31 in the EAX register will be clear when CPUID is called with an input parameter greater than highest value. All

other bit values returned by the processor in response to a CPUID instruction with EAX set to a value higher than appropriate for that processor are model specific and should not be relied upon.

3.1. Vendor ID String

In addition to returning the highest value in the EAX register, the Intel Vendor-ID string can be simultaneously verified as well. If the EAX register contains an input value of 0, the CPUID instruction also returns the vendor identification string in the EBX, EDX, and ECX registers (see Figure 2). These registers contain the ASCII string:

GenuineIntel

While any imitator of the Intel Architecture can provide the CPUID instruction, no imitator can legitimately claim that its part is a genuine Intel part. So the presence of the GenuineIntel string is an assurance that the CPUID instruction and the processor signature are implemented as described in this document. If the “GenuineIntel” string is not returned after execution of the CPUID instruction, do not rely upon the information described in this document to interpret the information returned by the CPUID instruction.

Table 1. Effects of EAX Contents on CPUID Instruction Output

Parameter	Outputs of CPUID
EAX = 0	EAX ← Highest value recognized by CPUID instruction
	EBX:EDX:ECX ← Vendor identification string
EAX = 1	EAX ← Processor signature, or Upper 32 bits of 96-bit processor serial number
	EDX ← Feature flags
	EBX[7:0] ← Brand ID EBX[31:8] ← Reserved
	ECX ← Reserved
EAX = 2	EAX:EBX:ECX:EDX ← Processor configuration parameters
EAX = 3	EDX:ECX ← lower 64-bits of 96-bit processor serial number
$4 \leq \text{EAX} \leq$ highest value	Reserved
EAX > highest value	EAX:EBX:ECX:EDX ← Undefined (Do not use.)

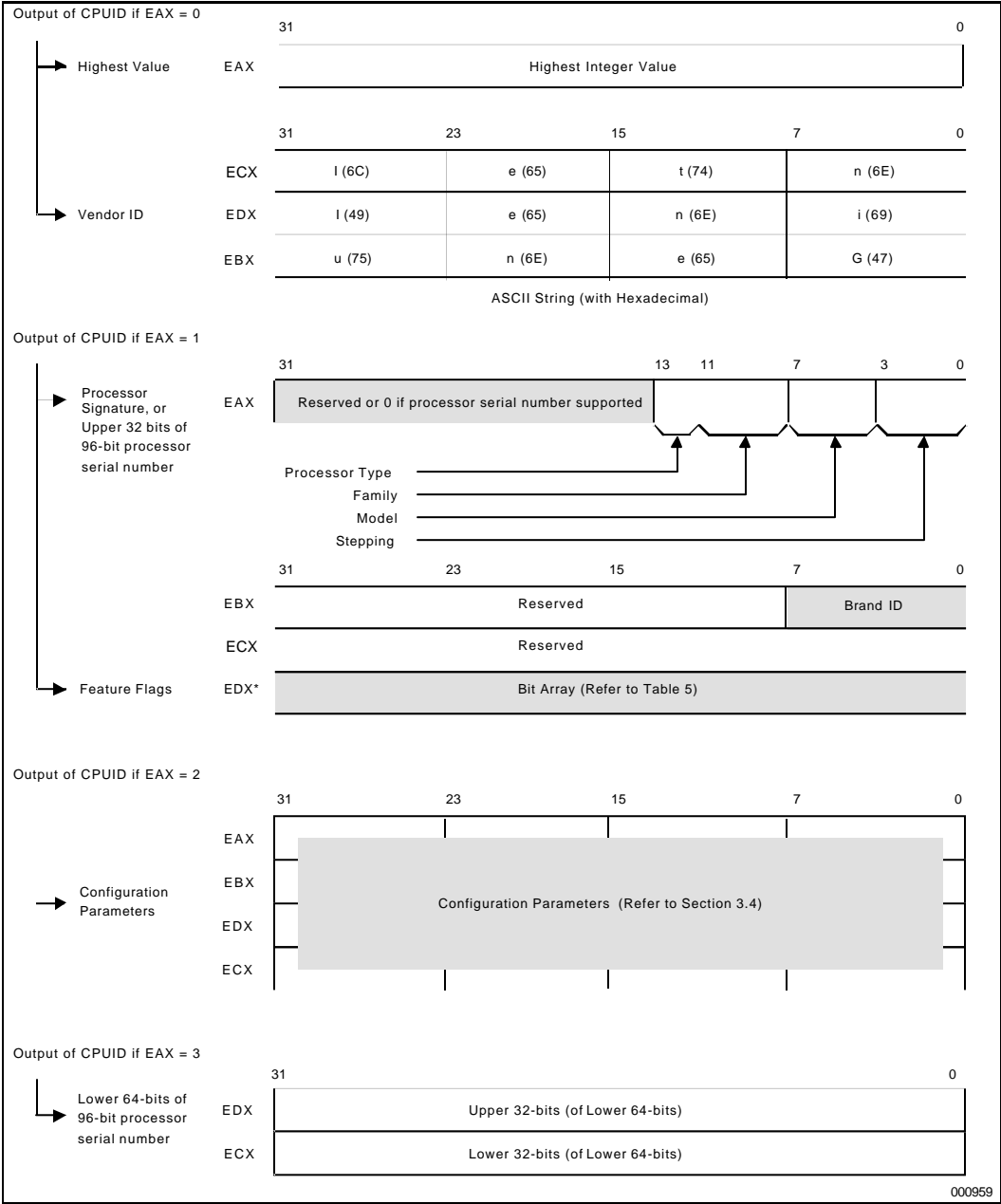


Figure 2. CPUID Instruction Outputs

3.2. Processor Signature

Beginning with the Intel486 processor family, the processor will return a processor identification signature value after reset in the EDX register (see Figure 3).

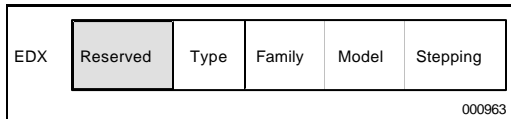


Figure 3. EDX Register after RESET

Processors that implement the CPUID instruction also return the processor identification signature after reset; however, the CPUID instruction gives you the flexibility of checking the processor signature at any time. Figure 3 shows the format of the signature for the Intel486, Pentium, Pentium Pro, Pentium II processors, Pentium II Xeon processors, Pentium II Overdrive processors, Intel Celeron processors, Pentium III processors, and Pentium III Xeon processors. Note that the EDX processor signature value after reset is equivalent to the processor signature output value in the EAX register in Figure 2. Table 3 shows the values returned in the EAX register currently defined for these processors.

The processor type, specified in bit positions 12 and 13 of Table 2 indicates whether the processor is an original OEM processor, an OverDrive processor, or a dual processor (capable of being used in a dual processor system). Table 2 shows the processor type values returned in bits 12 and 13 of the EAX register.

The family values, specified in bit positions 8 through 11, indicates whether the processor belongs to the Intel386, Intel486, Pentium or P6 family of processors. P6 family processors include all processors based on the Pentium® Pro processor architecture and have a family code equal to 6.

The model number, specified in bits 4 through 7, indicates the processor's family model number, while the stepping number in bits 0 through 3 indicates the revision number of that model.

The Pentium II processor, model 5, the Pentium II Xeon processor and the Intel Celeron processor, model 5 share the same family and model number. To differentiate between the processors, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If no L2 cache is returned, the processor is identified as an Intel Celeron processor, model 5. If

1M or 2M L2 cache size is reported, the processor is the Pentium II Xeon processor otherwise it is a Pentium II processor, model 5 or a Pentium II Xeon processor with 512K L2 cache.

The Pentium III processor, model 7, and the Pentium III Xeon processor, model 7, share the same family and model number. To differentiate between the processors, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If 1M or 2M L2 cache size is reported, the processor is the Pentium III Xeon processor otherwise it is a Pentium III processor or a Pentium III Xeon processor with 512K L2 cache.

Beginning with the Pentium III processor, model 8, the Pentium III Xeon processor, model 8, and the Intel Celeron processor, model 8, software should use the Brand ID values returned by the CPUID instruction when executed with EAX = 1 to determine the processor brand. Table 10 shows the processor brands defined by the Brand ID.

**Table 2. Processor Type
(Bit Positions 13 and 12)**

Value	Description
00	Original OEM processor
01	OverDrive® processor
10	Dual processor
11	Intel reserved (Do not use.)

Older versions of Intel486 SX, Intel486 DX and IntelDX2 processors do not support the CPUID instruction,² so they can only return the processor signature at reset. Refer to Table 3 to determine which processors support the CPUID instruction.

Figure 4 shows the format of the processor signature for Intel386 processors, which are different from other processors. Table 4 shows the values currently defined for these Intel386 processors.

Footnotes

² All Intel486 SL-enhanced and Write-Back enhanced processors are capable of executing the CPUID instruction. See Table 3.



Table 3. Intel486™, Pentium® Processor Family, OverDrive®, and P6 Family Processor Signatures

Type	Family	Model	Stepping	Description
00	0100	0000 and 0001	xxxx (1)	Intel486™ DX processors
00	0100	0010	xxxx (1)	Intel486 SX processors
00	0100	0011	xxxx (1)	Intel487™ processors
00	0100	0011	xxxx (1)	IntelDX2™ processors
00	0100	0011	xxxx (1)	IntelDX2 OverDrive® processors
00	0100	0100	xxxx (3)	Intel486 SL processor
00	0100	0101	xxxx (1)	IntelSX2™ processors
00	0100	0111	xxxx (3)	Write-Back Enhanced IntelDX2 processors
00	0100	1000	xxxx (3)	IntelDX4™ processors
00, 01	0100	1000	xxxx (3)	IntelDX4 OverDrive processors
00	0101	0001	xxxx (2)	Pentium® processors (60, 66)
00	0101	0010	xxxx (2)	Pentium processors (75, 90, 100, 120, 133, 150, 166, 200)
01 (4)	0101	0001	xxxx (2)	Pentium OverDrive processor for Pentium processor (60, 66)
01 (4)	0101	0010	xxxx (2)	Pentium OverDrive processor for Pentium processor (75, 90, 100, 120, 133)
01	0101	0011	xxxx (2)	Pentium OverDrive processors for Intel486 processor-based systems
00	0101	0100	xxxx (2)	Pentium processor with MMX™ technology (166, 200)
01	0101	0100	xxxx (2)	Pentium OverDrive processor with MMX™ technology for Pentium processor (75, 90, 100, 120, 133)
00	0110	0001	xxxx (2)	Pentium Pro processor
00	0110	0011	xxxx (2)	Pentium II processor, model 3
00	0110	0101(5)	xxxx (2)	Pentium II processor, model 5, Pentium II Xeon processor, model 5, and Intel Celeron processor, model 5
00	0110	0110	xxxx (2)	Intel Celeron processor, model 6
00	0110	0111(6)	xxxx (2)	Pentium III processor, model 7, and Pentium III Xeon processor, model 7
00	0110	1000(7)	xxxx(2)	Pentium III processor, model 8, Pentium III Xeon processor, model 8, and Intel Celeron processor, model 8
01	0110	0011	xxxx (2)	Intel Pentium II OverDrive processor



NOTES:

1. This processor does not implement the CPUID instruction.
2. Refer to the Intel486™ documentation, the Pentium® Processor Specification Update (Order Number 242480), the Pentium® Pro Processor Specification Update (Order Number 242689), the Pentium® II Processor Specification Update (Order Number 243337), the Pentium® II Xeon Processor Specification Update (Order Number 243776), the Intel Celeron Processor Specification Update (Order Number 243748), the Pentium® III Processor Specification Update (Order Number 244453) or the Pentium® III Xeon™ Processor Specification Update (Order Number 244460) for the latest list of stepping numbers.
3. Stepping 3 implements the CPUID instruction.
4. The definition of the type field for the OverDrive® processor is 01h. An erratum on the Pentium OverDrive processor will always return 00h as the type.
5. To differentiate between the Pentium II processor, model 5, Pentium II Xeon processor and the Intel Celeron processor, model 5, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If no L2 cache is returned, the processor is identified as an Intel Celeron processor, model 5. If 1M or 2M L2 cache size is reported, the processor is the Pentium II Xeon processor otherwise it is a Pentium II processor, model 5 or a Pentium II Xeon processor with 512K L2 cache size.
6. To differentiate between the Pentium III processor, model 7 and the Pentium III Xeon processor, model 7, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If 1M or 2M L2 cache size is reported, the processor is the Pentium III Xeon processor otherwise it is a Pentium III processor or a Pentium III Xeon processor with 512K L2 cache size.
7. To differentiate between the Pentium III processor, model 8 and the Pentium III Xeon processor, model 8, software should check the Brand ID values through executing CPUID instruction with EAX = 1.



3.3. Feature Flags

When the EAX register contains a value of 1, the CUID instruction (in addition to loading the processor signature in the EAX register) loads the EDX register with the feature flags. The current feature flags (when Flag = 1) indicate what features the processor supports. Table 5 lists the currently defined feature flag values.

For future processors, refer to the programmer's reference manual, user's manual, or the appropriate documentation for the latest feature flag values.

Use the feature flags in your applications to determine which processor features are supported. By using the CUID feature flags to predetermine processor features, your software can detect and avoid incompatibilities introduced by the addition or removal of processor features.

3.4. Cache Size and Format Information

When the EAX register contains a value of 2, the CUID instruction loads the EAX, EBX, ECX and EDX registers with descriptors that indicate the processors cache characteristics. The lower 8 bits of the EAX register (AL) contain a value that identifies the number of times the CUID has to be executed to obtain a complete image of the processor's caching systems. For example, the Pentium Pro processor returns a value of 1 in the lower 8 bits of the EAX register to indicate that the CUID instruction need only be executed once (with EAX = 2) to obtain a complete image of the processor configuration.

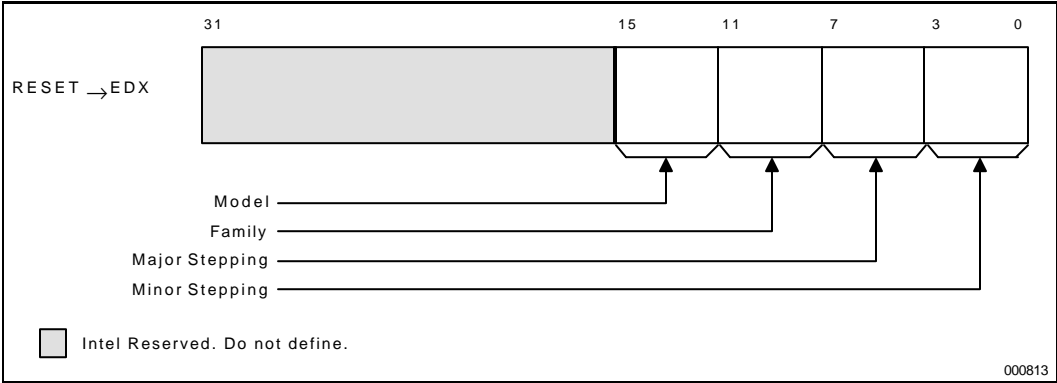


Figure 4. Processor Signature Format on Intel386™ Processors

Table 4. Intel386™ Processor Signatures

Type	Family	Major Stepping	Minor Stepping	Description
0000	0011	0000	xxxx	Intel386™ DX processor
0010	0011	0000	xxxx	Intel386 SX processor
0010	0011	0000	xxxx	Intel386 CX processor
0010	0011	0000	xxxx	Intel386 EX processor
0100	0011	0000 and 0001	xxxx	Intel386 SL processor
0000	0011	0100	xxxx	RapidCAD® coprocessor

Table 5. Feature Flag Values

Bit	Name	Description when Flag = 1	Comments
0	FPU	Floating-point unit on-chip	The processor contains an FPU that supports the Intel387 floating-point instruction set.
1	VME	Virtual Mode Extension	The processor supports extensions to virtual-8086 mode.
2	DE	Debugging Extension	The processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE	Page Size Extension	The processor supports 4-Mbyte pages.
4	TSC	Time Stamp Counter	The RDTSC instruction is supported including the CR4.TSD bit for access/privilege control.
5	MSR	Model Specific Registers	Model Specific Registers are implemented with the RDMSR, WRMSR instructions
6	PAE	Physical Address Extension	Physical addresses greater than 32 bits are supported.
7	MCE	Machine Check Exception	Machine Check Exception, Exception 18, and the CR4.MCE enable bit are supported
8	CX8	CMPXCHG8 Instruction Supported	The compare and exchange 8 bytes instruction is supported.
9	APIC	On-chip APIC Hardware Supported (1)	The processor contains a local APIC.
10		Reserved	Do not count on their value.
11	SEP	Fast System Call	Indicates whether the processor supports the Fast System Call instructions, SYSENTER and SYSEXIT. NOTE: Refer to Section 3.5 for further information regarding SYSENTER/ SYSEXIT feature and SEP feature bit.
12	MTRR	Memory Type Range Registers	The Processor supports the Memory Type Range Registers specifically the MTRR_CAP register.
13	PGE	Page Global Enable	The global bit in the PDEs and PTEs and the CR4.PGE enable bit are supported.
14	MCA	Machine Check Architecture	The Machine Check Architecture is supported, specifically the MCG_CAP register.
15	CMOV	Conditional Move Instruction Supported	The processor supports CMOVcc, and if the FPU feature flag (bit 0) is also set, supports the FCMOVCC and FCOMI instructions.
16	PAT	Page Attribute Table	Indicates whether the processor supports the Page Attribute Table. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on 4K granularity through a linear address.
17	PSE-36	36-bit Page Size Extension	Indicates whether the processor supports 4-Mbyte pages that are capable of addressing physical memory beyond 4GB. This feature indicates that the upper four bits of the physical address of the 4-Mbyte page is encoded by bits 13-16 of the page directory entry.
18	Processor serial number	Processor serial number is present and enabled	The processor supports the 96-bit processor serial number feature, and the feature is enabled.
19 – 22		Reserved	Do not count on their value.
23	MMX™ Technology	Intel Architecture MMX technology supported	The processor supports the MMX technology instruction set extensions to Intel Architecture.
24	FXSR	Fast floating point save	Indicates whether the processor supports the FXSAVE and



Table 5. Feature Flag Values

Bit	Name	Description when Flag = 1	Comments
		and restore	FXRSTOR instructions for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it uses the fast save/restore instructions.
25	Streaming SIMD Extension	Streaming SIMD Extensions supported	The processor supports the Streaming SIMD Extensions to the Intel Architecture.
26 – 31		Reserved	Do not count on their value.

8. NOTE:

9. 1. The processor contains a software-accessible Local APIC.

Fast System Call is supported

The remainder of the EAX register, and the EBX, ECX, and EDX registers, contain valid 8 bit descriptors. Table 6 shows that when bit 31 is zero, the register contains valid 8-bit descriptors. To decode descriptors, move sequentially from the most significant byte of the register down through the least significant byte of the register. Table 7 lists the current descriptor values and their respective cache characteristics. This list will be extended in the future as necessary.

Table 6. Descriptor Formats

Register bit 31	Descriptor Type	Description
1	Reserved	Reserved for future use.
0	8 bit descriptors	Descriptors point to a parameter table to identify cache characteristics. The descriptor is null if it has a 0 value.

3.5. SYSENTER/SYSEXIT – SEP Features Bit

The presence of this facility is indicated by the SYSENTER Present (SEP) bit 11 of CPUID. An operating system that detects the presence of the SEP bit must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present:

```
IF (CPUID SEP bit is set)
{
    IF (Family == 6) AND (Model Stepping < 0x33)
        Fast System Call is NOT supported
    ELSE
```

}

The Pentium Pro processor (Model = 1) returns a set SEP CPUID feature bit, but should not be used by software.

3.6. Pentium® Pro Processor Output Example

The Pentium Pro processor returns the values shown in Table 8. As the value of AL = 1, it is valid to interpret the remainder of the registers according to Table 7. Table 8 also shows that the MSB of the EAX register is 0. This indicates that the upper 8 bits constitute an 8-bit descriptor. The remaining register values in Table 8 show that the Pentium Pro processor has the following cache characteristics:

- A data TLB that maps 4K pages, is 4 way set associative, and has 64 entries.
- An instruction TLB that maps 4M pages, is fully associative, and has 2 entries.
- An instruction TLB that maps 4K pages, is 4 way set associative, and has 32 entries.
- An instruction cache that is 8K, is 4 way set associative, and has a 32 byte line size.
- A data TLB that maps 4M pages, is 4 way set associative, and has 8 entries.
- A data cache that is 8K, is 2 way set associative, and has a 32 byte line size.
- A unified cache that is 256K, is 4 way set associative, and has a 32 byte line size.



Table 7. Descriptor Decode Values

Descriptor Value	Cache Description
00h	Null
01h	Instruction TLB, 4K pages, 4-way set associative, 32 entries
02h	Instruction TLB, 4M pages, fully associative, 2 entries
03h	Data TLB, 4K pages, 4-way set associative, 64 entries
04h	Data TLB, 4M pages, 4-way set associative, 8 entries
06h	Instruction cache, 8K, 4-way set associative, 32 byte line size
08h	16KB instruction cache, 4-way set associative, 32 byte line size
0Ah	Data cache, 8K, 2-way set associative, 32 byte line size
0Ch	16KB data cache, 4-way set associative, 32 byte line size
40h	No L2 cache
41h	Unified cache, 32 byte cache line, 4-way set associative, 128K
42h	Unified cache, 32 byte cache line, 4-way set associative, 256K
43h	Unified cache, 32 byte cache line, 4-way set associative, 512K
44h	Unified cache, 32 byte cache line, 4-way set associative, 1M
45h	Unified cache, 32 byte cache line, 4-way set associative, 2M
82h	Unified cache, 32 byte cache line, 8 way set associative, 256K



Table 8. Pentium® Pro Processor, with 256K L2 Cache, CUID (EAX=2) Example Return Values

	31	23	15	7	0
EAX	03h	02h	01h	01h	
EBX	0	0	0	0	
ECX	0	0	0	0	
EDX	06h	04h	0Ah	42h	

3.7. Pentium® II Processor, model 3 Output Example

The Pentium II processor, model 3 returns the values shown in Table 9. If the value of AL=1, it is valid to interpret the remainder of the registers according to Table 7. Table 9 also shows the MSB of EAX register is 0. As with the Pentium Pro processor this indicates the upper 8 bits constitute an 8-bit descriptor. The remaining register values in Table 9 shows the Pentium II processor has the following cache characteristics:

- A data TLB that maps 4K pages, is 4 way set associative, and has 64 entries.
- An instruction TLB that maps 4M pages, is fully associative, and has 2 entries.
- An instruction TLB that maps 4K pages, is 4 way set associative, and has 32 entries.
- A data cache that is 16K, is 4 way set associative, and has a 32 byte line size.
- A data TLB that maps 4M pages, is 4 way set associative, and has 8 entries.
- An instruction cache that is 16K, is 4 way set associative, and has a 32 byte line size.
- A unified cache that is 512K, is 4 way set associative, and has a 32 byte line size.

4.0. PROCESSOR SERIAL NUMBER

The Pentium III processors and the Pentium III Xeon processors extend the concept of processor identification with the addition of processor serial number. Processor serial number is a 96-bit number accessible through the CUID instruction. Processor serial number can be used by applications to identify a processor, and by extension, its system.

The processor serial number creates a software accessible identity for an individual processor. The processor serial number, combined with other qualifiers, could be applied to user identification. Applications include membership authentication, data backup/restore protection, removable storage data protection, managed access to files, or to confirm document exchange between appropriate users.

Processor serial number is another tool for use in asset management, product tracking, remote systems load and configuration, or to aid in boot-up configuration. In the case of system service, processor serial number could be used to differentiate users during help desk access, or track error reporting.

Table 9. Pentium® II Processor, model 3 with 512K L2 Cache, CUID (EAX=2) Example Return Values

	31	23	15	7	0
EAX	03h	02h	01h	01h	
EBX	0	0	0	0	
ECX	0	0	0	0	
EDX	0Ch	04h	08h	43h	



Processor serial number provides an identifier for the processor, but should not be assumed to be unique in itself. There are potential modes in which erroneous processor serial numbers may be reported. For example, in the event a processor is operated outside its recommended operating specifications, (e.g. voltage, frequency, etc.) the processor serial number may not be correctly read from the processor. Improper BIOS or software operations could yield an inaccurate processor serial number. These events could lead to possible erroneous or duplicate processor serial numbers being reported. System manufacturers can strengthen the robustness of the feature by including redundancy features, or other fault tolerant methods.

Processor serial number used as a qualifier for another independent number could be used to create an electrically accessible number that is likely to be distinct. Processor serial number is one building block useful for the purpose of enabling the trusted, connected PC.

4.1. Presence of Processor Serial Number

To determine if the processor serial number feature is supported, the program should set the EAX register parameter value to “1” and then execute the CPUID instruction as follows:

```
MOV EAX, 01H
CPUID
```

After execution of the CPUID instruction, the EDX register contains the Feature Flags. If Feature Flags bit 18 equals “1”, then the processor serial number feature is supported, and enabled. If Feature Flags bit 18 equals “0”, the processor serial number feature is either not supported, or disabled.

4.2. Forming the 96-bit Processor Serial Number

The 96-bit processor serial number is the concatenation of three 32-bit entities.

To access the most significant 32-bits of the processor serial number the program should set the EAX register parameter value to “1” and then execute the CPUID instruction as follows:

```
MOV EAX, 01H
CPUID
```

After execution of the CPUID instruction, the EAX register contains the Processor Signature. The Processor Signature comprises the most significant 32-bits of the

processor serial number. The value in EAX should be saved prior to gathering the remaining 64-bits of the processor serial number.

To access the remaining 64-bits of the processor serial number the program should set the EAX register parameter value to “3” and then execute the CPUID instruction as follows:

```
MOV EAX, 03H
CPUID
```

After execution of the CPUID instruction, the EDX register contains the middle 32-bits, and the ECX register contains the least significant 32-bits of the processor serial number. Software may then concatenate the saved Processor Signature, EDX, and ECX before returning the complete 96-bit processor serial number.

Processor serial number should be displayed as 6 groups of 4 hex nibbles (Ex. XXXX-XXXX-XXXX-XXXX-XXXX-XXXX where X represents a hex digit). Alpha hex characters should be displayed as capital letters.

5.0. BRAND ID

Beginning with the Pentium III processors, model 8, the Pentium III Xeon processors, model 8, and Intel Celeron processor, model 8, the concept of processor identification is further extended with the addition of Brand ID. Brand ID is an 8-bit number accessible through the CPUID instruction. Brand ID is used by applications to accurately identify the processor.

Processors that implement the Brand ID feature return the Brand ID in bits 7 through 0 of the EBX register when the CPUID instruction is executed with EAX=1 (see Table 10). Processors that do not support the feature return a value of 0 in EBX bits 7 through 0.

To differentiate previous models of the Pentium II processor, Pentium II Xeon processor, Intel Celeron processor, Pentium III processor and Pentium III Xeon processor, application software relied on the L2 cache descriptors. In a few cases the results were ambiguous, for example software could not accurately differentiate a Pentium II processor from a Pentium II Xeon processor with a 512K L2 cache. Brand ID eliminates this ambiguity by providing a software accessible value unique to each processor brand. Table 10 shows the values defined for each processor.



Table 10. Brand ID, CPUID (EAX=1) Return Values (bits 7 through 0)

Value	Description
0	Unsupported
1	Intel® Celeron™ processor
2	Intel® Pentium® III processor
3	Intel® Pentium® III Xeon™ processor
4h - FFh	Reserved

6.0. USAGE GUIDELINES

This document presents Intel-recommended feature-detection methods. Software should not try to identify features by exploiting programming tricks, undocumented features, or otherwise deviating from the guidelines presented in this application note.

The following guidelines are intended to help programmers maintain the widest range of compatibility for their software.

- Do not depend on the absence of an invalid opcode trap on the CPUID opcode to detect the CPUID instruction. Do not depend on the absence of an invalid opcode trap on the PUSHFD opcode to detect a 32-bit processor. Test the ID flag, as described in Section 2.0. and shown in Section 7.0.
- Do not assume that a given family or model has any specific feature. For example, do not assume the family value 5 (Pentium processor) means there is a floating-point unit on-chip. Use the feature flags for this determination.
- Do not assume processors with higher family or model numbers have all the features of a processor with a lower family or model number. For example, a processor with a family value of 6 (P6 family processor) may not necessarily have all the features of a processor with a family value of 5.
- Do not assume that the features in the OverDrive processors are the same as those in the OEM version of the processor. Internal caches and instruction execution might vary.
- Do not use undocumented features of a processor to identify steppings or features. For example, the Intel386 processor A-step had bit instructions that were withdrawn with the B-step. Some software attempted to execute these instructions and depended on the invalid-opcode exception as a

signal that it was not running on the A-step part. The software failed to work correctly when the Intel486 processor used the same opcodes for different instructions. The software should have used the stepping information in the processor signature.

- Test feature flags individually and do not make assumptions about undefined bits. For example, it would be a mistake to test the FPU bit by comparing the feature register to a binary 1 with a compare instruction.
- Do not assume the clock of a given family or model runs at a specific frequency, and do not write processor speed-dependent code, such as timing loops. For instance, an OverDrive Processor could operate at a higher internal frequency and still report the same family and/or model. Instead, use a combination of the system's timers to measure elapsed time and the TSC (Time Stamp Counter) to measure processor core clocks to allow direct calibration of the processor core.
- Processor model-specific registers may differ among processors, including in various models of the Pentium processor. Do not use these registers unless identified for the installed processor. This is particularly important for systems upgradeable with an OverDrive processor. Only use Model Specific registers that are defined in the BIOS writers guide for that processor.
- Do not rely on the result of the CPUID algorithm when executed in virtual 8086 mode.
- Do not assume any ordering of model and/or stepping numbers. They are assigned arbitrarily.
- Do not assume processor serial number is a unique number without further qualifiers.
- Display processor serial number as 6 groups of 4 hex nibbles (Ex. XXXX-XXXX-XXXX-XXXX-XXXX-XXXX where X represents a hex digit).
- Display alpha hex characters as capital letters.
- A zero processor serial number is invalid or indicates that processor serial number is invalid on this processor.

7.0. PROPER IDENTIFICATION SEQUENCE

The cpuid3a.asm program example demonstrates the correct use of the CPUID instruction (see Example 1). It also shows how to identify earlier processor generations that do not implement the processor signature or CPUID



instruction (see Figure 5). This program example contains the following two procedures:

- `get_cpu_type` identifies the processor type. Figure 5 illustrates the flow of this procedure.
- `get_fpu_type` determines the type of floating-point unit (FPU) or math coprocessor (MCP).

This procedure has been tested with 8086, 80286, Intel386, Intel486, Pentium processor, Pentium processor

with MMX technology, OverDrive processor with MMX technology, Pentium Pro processors, Pentium II processors, Pentium II Xeon processors, Pentium II Overdrive processors, Intel Celeron processors, Pentium III processors, and Pentium III Xeon processors. This program example is written in assembly language and is suitable for inclusion in a run-time library, or as system calls in operating systems.

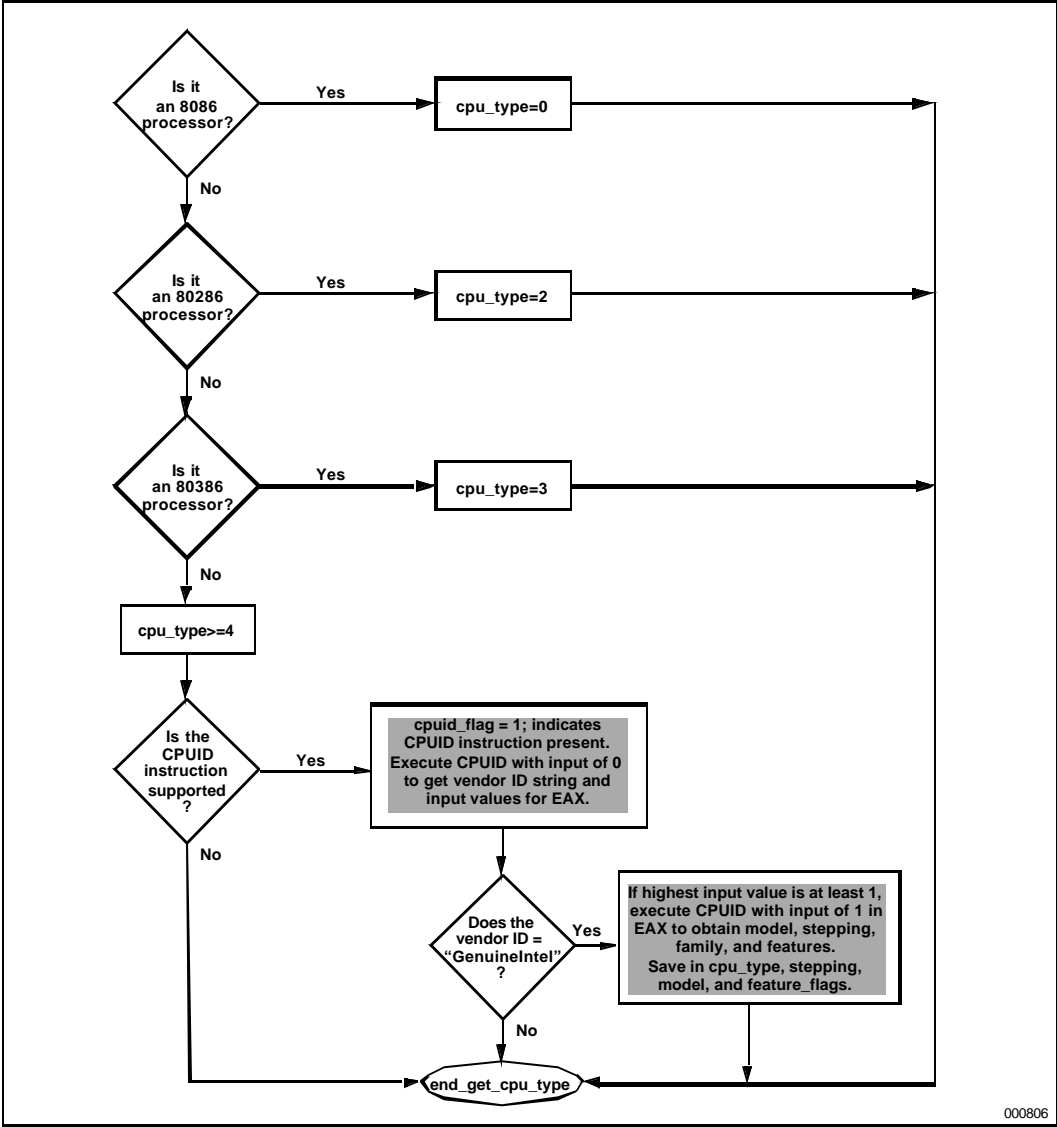


Figure 5. Flow of Processor get_cpu_type Procedure

8.0. USAGE PROGRAM EXAMPLES

The cpuid3b.asm or cpuid3.c program examples demonstrate applications that call get_cpu_type and get_fpu_type procedures and interpret the returned information. This code is shown in Example 2 and Example 3. The results, which are displayed on the

monitor, identify the installed processor and features. The cpuid3b.asm example is written in assembly language and demonstrates an application that displays the returned information in the DOS environment. The cpuid3.c example is written in the C language (see Example 2 and Example 3). Figure 6 presents an

overview of the relationship between the three program examples.

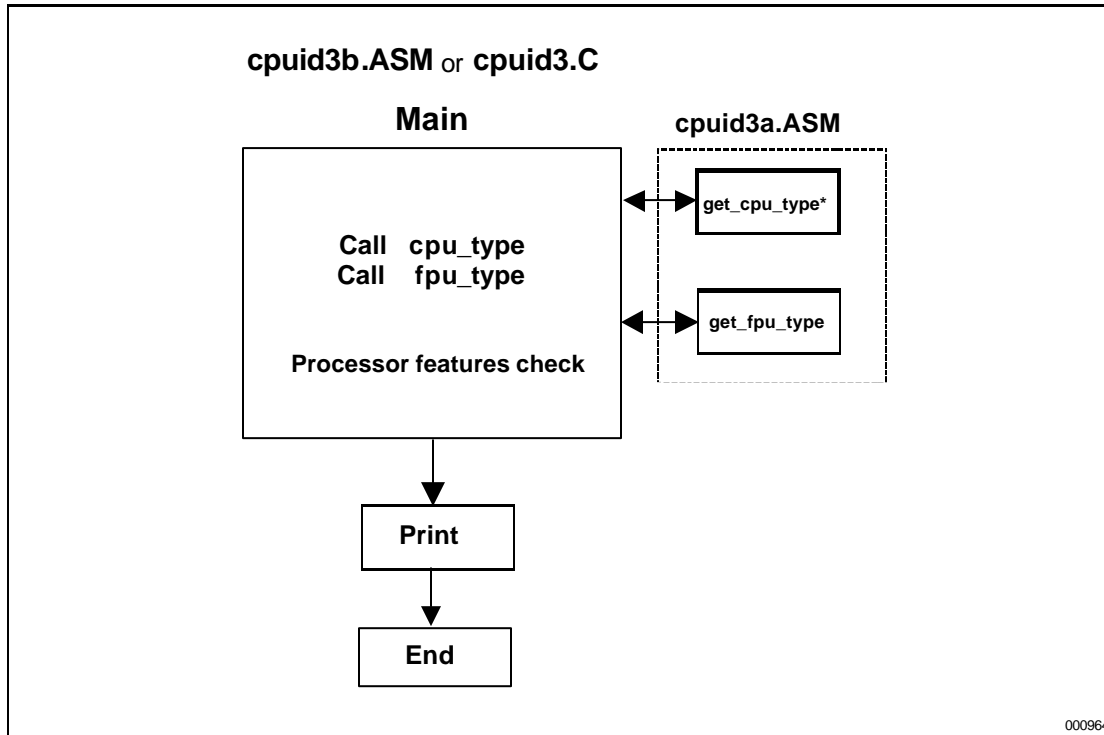


Figure 6. Flow of Processor Identification Extraction Procedure



Example 1. Processor Identification Extraction Procedure

```

; Filename: cpuid3a.asm
; Copyright 1993, 1994, 1995, 1996, 1997, 1998 by Intel Corp.
;
; This program has been developed by Intel Corporation. Intel
; has various intellectual property rights which it may assert
; under certain circumstances, such as if another
; manufacturer's processor mis-identifies itself as being
; "GenuineIntel" when the CPUID instruction is executed.
;
; Intel specifically disclaims all warranties, express or
; implied, and all liability, including consequential and other
; indirect damages, for the use of this program, including
; liability for infringement of any proprietary rights,
; and including the warranties of merchantability and fitness
; for a particular purpose. Intel does not assume any
; responsibility for any errors which may appear in this program
; nor any responsibility to update it.
;
; This code contains two procedures:
; _get_cpu_type: Identifies processor type in _cpu_type:
;               0=8086/8088 processor
;               2=Intel 286 processor
;               3=Intel386™ family processor
;               4=Intel486™ family processor
;               5=Pentium® family processor
;               6=P6 family of processors
;
; _get_fpu_type: Identifies FPU type in _fpu_type:
;               0=FPU not present
;               1=FPU present
;               2=287 present (only if _cpu_type=3)
;               3=387 present (only if _cpu_type=3)
;
; This program has been tested with the Microsoft Developer Studio.
; This code correctly detects the current Intel 8086/8088,
; 80286, 80386, 80486, Pentium(R) processor, Pentium(R) Pro
; processor, Pentium(R) II processor, Pentium II Xeon(TM) processor,
; Pentium II Overdrive(R), Intel Celeron processor, Pentium III processor
; and Pentium III Xeon processor in the real-address mode only.
;
; NOTE: When using this code with C program cpuid3.c, 32-bit
; segments are recommended.
;
; To assemble this code with TASM, add the JUMPS directive.
; jumps ; Uncomment this line for TASM
;
; TITLE cpuid3a
;
; comment this line for 32-bit segments
;
DOSSEG

```



```
;
;      uncomment the following 2 lines for 32-bit segments
;
;      .386
;      .model    flat
;
;      comment this line for 32-bit segments
;
;      .model    small

CPU_ID  MACRO
db      0fh                ; Hardcoded CPUID instruction
db      0a2h
ENDM

.data
public  _cpu_type
public  _fpu_type
public  _v86_flag
public  _cpuid_flag
public  _intel_CPU
public  _vendor_id
public  _cpu_signature
public  _features_ecx
public  _features_edx
public  _features_ebx
public  _cache_eax
public  _cache_ebx
public  _cache_ecx
public  _cache_edx
public  _sep_flag
public  _platform_id

_cpu_type      db      0
_fpu_type      db      0
_v86_flag      db      0
_cpuid_flag     db      0
_intel_CPU     db      0
_sep_flag      db      0
_vendor_id     db      "-----"
intel_id       db      "GenuineIntel"
_cpu_signature  dd      0
_features_ecx   dd      0
_features_edx   dd      0
_features_ebx   dd      0
_cache_eax     dd      0
_cache_ebx     dd      0
_cache_ecx     dd      0
_cache_edx     dd      0
fp_status      dw      0
_platform_id    db      0

.code
;
;      comment this line for 32-bit segments
;
```

```

.8086
;
;      uncomment this line for 32-bit segments
;
;      .386

*****
;
;      public      _get_cpu_type
;      _get_cpu_type      proc
;
;      This procedure determines the type of processor in a system
;      and sets the _cpu_type variable with the appropriate
;      value. If the CUID instruction is available, it is used
;      to determine more specific details about the processor.
;      All registers are used by this procedure, none are preserved.
;      To avoid AC faults, the AM bit in CR0 must not be set.
;
;      Intel 8086 processor check
;      Bits 12-15 of the FLAGS register are always set on the
;      8086 processor.
;
;
;      For 32-bit segments comment the following lines down to the next
;      comment line that says "STOP"
;
check_8086:
    pushf                ; push original FLAGS
    pop      ax           ; get original FLAGS
    mov      cx, ax       ; save original FLAGS
    and      ax, 0fffh    ; clear bits 12-15 in FLAGS
    push     ax           ; save new FLAGS value on stack
    popf         ; replace current FLAGS value
    pushf         ; get new FLAGS
    pop      ax           ; store new FLAGS in AX
    and      ax, 0f000h   ; if bits 12-15 are set, then
    cmp      ax, 0f000h   ; processor is an 8086/8088
    mov      _cpu_type, 0 ; turn on 8086/8088 flag
    jne      check_80286  ; go check for 80286
    push     sp           ; double check with push sp
    pop      dx           ; if value pushed was different
    cmp      dx, sp       ; means it's really an 8086
    jne      end_cpu_type ; jump if processor is 8086/8088
    mov      _cpu_type, 10h ; indicate unknown processor
    jmp      end_cpu_type

;
;      Intel 286 processor check
;      Bits 12-15 of the FLAGS register are always clear on the
;      Intel 286 processor in real-address mode.

.286
check_80286:
    smsw     ax           ; save machine status word
    and      ax, 1        ; isolate PE bit of MSW
    mov      _v86_flag, al ; save PE bit to indicate V86

    or       cx, 0f000h   ; try to set bits 12-15

```



```

push    cx                ; save new FLAGS value on stack
popf                    ; replace current FLAGS value
pushf                    ; get new FLAGS
pop     ax                ; store new FLAGS in AX
and     ax, 0f000h        ; if bits 12-15 are clear
mov     _cpu_type, 2      ; processor=80286, turn on 80286 flag
jz      end_cpu_type      ; jump if processor is 80286

;
; Intel386 processor check
; The AC bit, bit #18, is a new bit introduced in the EFLAGS
; register on the Intel486 processor to generate alignment
; faults.
; This bit cannot be set on the Intel386 processor.

.386
;
; "STOP"
;
;
; it is safe to use 386 instructions
check_80386:
pushfd                    ; push original EFLAGS
pop     eax               ; get original EFLAGS
mov     ecx, eax          ; save original EFLAGS
xor     eax, 40000h       ; flip AC bit in EFLAGS
push    eax              ; save new EFLAGS value on stack
popfd                    ; replace current EFLAGS value
pushfd                    ; get new EFLAGS
pop     eax               ; store new EFLAGS in EAX
xor     eax, ecx          ; can't toggle AC bit, processor=80386
mov     _cpu_type, 3      ; turn on 80386 processor flag
jz      end_cpu_type      ; jump if 80386 processor
push    ecx
popfd                    ; restore AC bit in EFLAGS first

;
; Intel486 processor check
; Checking for ability to set/clear ID flag (Bit 21) in EFLAGS
; which indicates the presence of a processor with the CUID
; instruction.

.486
check_80486:
mov     _cpu_type, 4      ; turn on 80486 processor flag
mov     eax, ecx          ; get original EFLAGS
xor     eax, 200000h      ; flip ID bit in EFLAGS
push    eax              ; save new EFLAGS value on stack
popfd                    ; replace current EFLAGS value
pushfd                    ; get new EFLAGS
pop     eax               ; store new EFLAGS in EAX
xor     eax, ecx          ; can't toggle ID bit,
je      end_cpu_type      ; processor=80486

;
; Execute CUID instruction to determine vendor, family,
; model, stepping and features. For the purpose of this
; code, only the initial set of CUID information is saved.

mov     _cpuid_flag, 1    ; flag indicating use of CUID inst.

```



```

push    ebx                                ; save registers
push    esi
push    edi
mov     eax, 0                            ; set up for CUID instruction
CPU_ID                                     ; get and save vendor ID

mov     dword ptr _vendor_id, ebx
mov     dword ptr _vendor_id[+4], edx
mov     dword ptr _vendor_id[+8], ecx

cmp     dword ptr intel_id, ebx
jne     end_cpuid_type
cmp     dword ptr intel_id[+4], edx
jne     end_cpuid_type
cmp     dword ptr intel_id[+8], ecx
jne     end_cpuid_type                    ; if not equal, not an Intel processor

mov     _intel_CPU, 1                     ; indicate an Intel processor
cmp     eax, 1                           ; make sure 1 is valid input for CUID
jl      end_cpuid_type                   ; if not, jump to end
mov     CPU_ID, 1                         ; get family/model/stepping/features
mov     _cpu_signature, eax
mov     _features_ebx, ebx
mov     _features_edx, edx
mov     _features_ecx, ecx

shr     eax, 8                           ; isolate family
and     eax, 0fh
mov     _cpu_type, al                     ; set _cpu_type with family

; Execute CUID instruction to determine the cache descriptor
; information.

mov     eax, 0                           ; set up to check the EAX value
CPU_ID
cmp     ax, 2                             ; Are cache descriptors supported?
jl      end_cpuid_type

mov     eax, 2                           ; set up to read cache descriptor
CPU_ID
cmp     al, 1                             ; Is one iteration enough to obtain
jne     end_cpuid_type                   ; cache information?
                                           ; This code supports one iteration
                                           ; only.
mov     _cache_eax, eax                   ; store cache information
mov     _cache_ebx, ebx                   ; NOTE: for future processors, CUID
mov     _cache_ecx, ecx                   ; instruction may need to be run more
mov     _cache_edx, edx                   ; than once to get complete cache
                                           ; information

end_cpuid_type:
pop     edi                               ; restore registers
pop     esi
pop     ebx

;
; comment this line for 32-bit segments

```

```

;
.8086
end_cpu_type:
    ret
_get_cpu_type    endp

;*****
;

    public    _get_fpu_type
    _get_fpu_type    proc

;
;   This procedure determines the type of FPU in a system
;   and sets the _fpu_type variable with the appropriate value.
;   All registers are used by this procedure, none are preserved.
;
;   Coprocessor check
;   The algorithm is to determine whether the floating-point
;   status and control words are present. If not, no
;   coprocessor exists. If the status and control words can
;   be saved, the correct coprocessor is then determined
;   depending on the processor type. The Intel386 processor can
;   work with either an Intel287 NDP or an Intel387 NDP.
;   The infinity of the coprocessor must be checked to determine
;   the correct coprocessor type.
;
    fninit                ; reset FP status word
    mov     fp_status,5a5ah ; initialize temp word to non-zero
    fnstsw   fp_status      ; save FP status word
    mov     ax,fp_status    ; check FP status word
    cmp     al,0            ; was correct status written
    mov     _fpu_type,0     ; no FPU present
    jne     end_fpu_type

check_control_word:
    fnstcw   fp_status      ; save FP control word
    mov     ax,fp_status    ; check FP control word
    and     ax,103fh        ; selected parts to examine
    cmp     ax,3fh          ; was control word correct
    mov     _fpu_type,0
    jne     end_fpu_type    ; incorrect control word, no FPU
    mov     _fpu_type,1

;
;   80287/80387 check for the Intel386 processor

check_infinity:
    cmp     _cpu_type,3
    jne     end_fpu_type
    fldl
    fldz
    fdiv
    fld     st
    fchs
    fcompp
    fstsw   fp_status
    mov     ax,fp_status
    mov     _fpu_type,2     ; store Intel287 NDP for FPU type
; must use default control from FNINIT
; form infinity
; 8087/Intel287 NDP say +inf = -inf
; form negative infinity
; Intel387 NDP says +inf <> -inf
; see if they are the same
; look at status from FCOMPP

```



```

        sahf
        jz     end_fpu_type      ; see if infinities matched
        mov    _fpu_type, 3     ; jump if 8087 or Intel287 is present
end_fpu_type:
        ret                    ; store Intel387 NDP for FPU type
_get_fpu_type    endp
end
```



Example 2. Processor Identification Procedure in Assembly Language

```

;      Filename: cpuid3b.asm
;      Copyright 1993, 1994, 1995, 1996, 1997, 1998 by Intel Corp.
;
;      This program has been developed by Intel Corporation. Intel
;      has various intellectual property rights which it may assert
;      under certain circumstances, such as if another
;      manufacturer's processor mis-identifies itself as being
;      "GenuineIntel" when the CPUID instruction is executed.
;
;      Intel specifically disclaims all warranties, express or
;      implied, and all liability, including consequential and
;      other indirect damages, for the use of this program,
;      including liability for infringement of any proprietary
;      rights, and including the warranties of merchantability and
;      fitness for a particular purpose. Intel does not assume any
;      responsibility for any errors which may appear in this
;      program nor any responsibility to update it.
;
;      This program contains three parts:
;      Part 1:   Identifies processor type in the variable
;                _cpu_type:
;
;      Part 2:   Identifies FPU type in the variable _fpu_type:
;
;      Part 3:   Prints out the appropriate message. This part is
;                specific to the DOS environment and uses the DOS
;                system calls to print out the messages.
;
;      This program has been tested with the Microsoft Developer Studio. If
;      this code is assembled with no options specified and linked
;      with the cpuid3a module, it correctly identifies the current
;      Intel 8086/8088, 80286, 80386, 80486, Pentium®, Pentium® Pro,
;      Pentium® II processors, Pentium® II Xeon processors, Pentium II Overdrive
;      processors, Intel Celeron™ processors, Pentium III processors and Pentium III Xeon processors
;      in the real-address mode.
;
; NOTE: This code is written using 16-bit Segments
;
;      To assemble this code with TASM, add the JUMPS directive.
;      jumps                                ; Uncomment this line for TASM
;
;      TITLE    cpuid3b
;
DOSSEG
.model    small

.stack    100h

OP_O      MACRO
db        66h                                ; hardcoded operand override
ENDM

.data

```



```

extrn      _cpu_type:      byte
extrn      _fpu_type:      byte
extrn      _cpuid_flag:    byte
extrn      _intel_CPU:     byte
extrn      _vendor_id:     byte
extrn      _sep_flag:      byte
extrn      _cpu_signature: dword
extrn      _features_ecx:  dword
extrn      _features_edx:  dword
extrn      _features_ebx:  dword
extern     _cache_eax:     dword
extern     _cache_ebx:     dword
extern     _cache_ecx:     dword
extern     _cache_edx:     dword
extrn      _platform_id:   byte

;      The purpose of this code is to identify the processor and
;      coprocessor that is currently in the system. The program
;      first determines the processor type. Then it determines
;      whether a coprocessor exists in the system. If a
;      coprocessor or integrated coprocessor exists, the program
;      identifies the coprocessor type. The program then prints
;      the processor and floating point processors present and type.

.code
.8086
start:

    mov     ax, @data
    mov     ds, ax                ; set segment register
    mov     es, ax                ; set segment register
    and     sp, not 3             ; align stack to avoid AC fault
    call    _get_cpu_type         ; determine processor type
    call    _get_fpu_type
    call    print

    mov     ax, 4c00h
    int     21h

;*****

    extrn   _get_cpu_type: proc

;*****

    extrn   _get_fpu_type: proc

;*****

FPU_FLAG      equ 0001h
VME_FLAG      equ 0002h
DE_FLAG       equ 0004h
PSE_FLAG      equ 0008h
TSC_FLAG      equ 0010h
MSR_FLAG      equ 0020h
PAE_FLAG      equ 0040h
MCE_FLAG      equ 0080h
30

```



```
CX8_FLAG          equ 0100h
APIC_FLAG          equ 0200h
SEP_FLAG           equ 0800h
MTRR_FLAG          equ 1000h
PGE_FLAG           equ 2000h
MCA_FLAG           equ 4000h
CMOV_FLAG          equ 8000h
PAT_FLAG           equ 10000h
PSE36_FLAG         equ 20000h
PSNUM_FLAG         equ 40000h
MMX_FLAG           equ 800000h
FXSR_FLAG          equ 1000000h
SIMD_FLAG          equ 2000000h

.data
id_msg             db      "This system has a$"
cp_error           db      "n unknown processor$"
cp_8086            db      "n 8086/8088 processor$"
cp_286             db      "n 80286 processor$"
cp_386            db      "n 80386 processor$"

cp_486             db      "n 80486DX, 80486DX2 processor or"
                  db      " 80487SX math coprocessor$"
cp_486sx           db      "n 80486SX processor$"

fp_8087            db      " and an 8087 math coprocessor$"
fp_287             db      " and an 80287 math coprocessor$"
fp_387            db      " and an 80387 math coprocessor$"

intel486_msg       db      " Genuine Intel486(TM) processor$"
intel486dx_msg     db      " Genuine Intel486(TM) DX processor$"
intel486sx_msg     db      " Genuine Intel486(TM) SX processor$"
inteldx2_msg       db      " Genuine IntelDX2(TM) processor$"
intelsx2_msg       db      " Genuine IntelSX2(TM) processor$"
inteldx4_msg       db      " Genuine IntelDX4(TM) processor$"
inteldx2wb_msg     db      " Genuine Write-Back Enhanced"
                  db      " IntelDX2(TM) processor$"
pentium_msg        db      " Genuine Intel Pentium(R) processor$"
pentiumpro_msg     db      " Genuine Intel Pentium(R) Pro processor$"

pentiumiimodel3_msg db      " Genuine Intel Pentium(R) II processor, model 3$"
pentiumiixeon_m5_msg db      " Genuine Intel Pentium(R) II processor, model 5 or Intel Pentium(R) II Xeon(TM)
processor$"
pentiumiixeon_msg  db      " Genuine Intel Pentium(R) II Xeon(TM) processor$"
celeron_msg        db      " Genuine Intel Celeron(TM) processor, model 5$"
celeronmodel6_msg db      " Genuine Intel Celeron(TM) processor, model 6$"
celeron_brand      db      " Genuine Intel Celeron(TM) processor$"
pentiumiii_msg     db      " Genuine Intel Pentium(R) III processor, model 7 or Intel Pentium(R) III Xeon(TM)
processor, model 7$"
pentiumiixeon_msg  db      " Genuine Intel Pentium(R) III Xeon(TM) processor, model 7$"
pentiumiixeon_brand db      " Genuine Intel Pentium(R) III Xeon(TM) processor$"
pentiumiii_brand   db      " Genuine Intel Pentium(R) III processor$"
unknown_msg        db      "n unknown Genuine Intel processor$"

brand_entry        struct
    brand_value     db      ?
```





```
brand_string      dw      ?
brand_entry       ends

brand_table       brand_entry    <1, offset celeron_brand>
                  brand_entry    <2, offset pentiumiii_brand>
                  brand_entry    <3, offset pentiumiiiixon_brand>

brand_table_size  equ      ($ - offset brand_table) / (sizeof brand_entry)
```

; The following 16 entries must stay intact as an array

```
intel_486_0      dw      offset intel486dx_msg
intel_486_1      dw      offset intel486dx_msg
intel_486_2      dw      offset intel486sx_msg
intel_486_3      dw      offset inteldx2_msg
intel_486_4      dw      offset intel486_msg
intel_486_5      dw      offset intelsx2_msg
intel_486_6      dw      offset intel4 86_msg
intel_486_7      dw      offset inteldx2wb_msg
intel_486_8      dw      offset inteldx4_msg
intel_486_9      dw      offset intel486_msg
intel_486_a      dw      offset intel486_msg
intel_486_b      dw      offset intel486_msg
intel_486_c      dw      offset intel486_msg
intel_486_d      dw      offset intel486_msg
intel_486_e      dw      offset intel486_msg
intel_486_f      dw      offset intel486_msg
```

; end of array

```
family_msg      db      13,10,"Processor Family: $"
model_msg       db      13,10,"Model:      $"
stepping_msg    db      13,10,"Stepping:   "
cr_1f           db      13,10,"$"
turbo_msg       db      13,10,"The processor is an OverDrive(R)"
                db      " processor$"
dp_msg          db      13,10,"The processor is the upgrade"
                db      " processor in a dual processor system$"
fpu_msg         db      13,10,"The processor contains an on-chip"
                db      " FPU$"
vme_msg         db      13,10,"The processor supports Virtual"
                db      " Mode Extensions$"
de_msg          db      13,10,"The processor supports Debugging"
                db      " Extensions$"
pse_msg         db      13,10,"The processor supports Page Size"
                db      " Extensions$"
tsc_msg         db      13,10,"The processor supports Time Stamp"
                db      " Counter$"
msr_msg         db      13,10,"The processor supports Model"
                db      " Specific Registers$"
pae_msg         db      13,10,"The processor supports Physical"
                db      " Address Extensions$"
mce_msg         db      13,10,"The processor supports Machine"
                db      " Check Exceptions$"
cx8_msg         db      13,10,"The processor supports the"
                db      " CMPXCHG8B instruction$"
apic_msg        db      13,10,"The processor contains an on-chip"
                db      " APIC$"
```





```
sep_msg      db      13,10,"The processor supports Fast System"
              db      " Call$"
no_sep_msg   db      13,10,"The processor does not support Fast"
              db      " System Call$"
mtrr_msg     db      13,10,"The processor supports Memory Type"
              db      " Range Registers$"
pge_msg      db      13,10,"The processor supports Page Global"
              db      " Enable$"
mca_msg      db      13,10,"The processor supports Machine"
              db      " Check Architecture$"
cmov_msg     db      13,10,"The processor supports Conditional"
              db      " Move Instruction$"
pat_msg      db      13,10,"The processor supports Page Attribute"
              db      " Table$"
pse36_msg    db      13,10,"The processor supports 36-bit Page"
              db      " Size Extension$"
psnum_msg    db      13,10,"The processor supports the"
              db      " processor serial number$"
mmx_msg      db      13,10,"The processor supports Intel Architecture"
              db      " MMX(TM) Technology$"
fxsr_msg     db      13,10,"The processor supports Fast floating point"
              db      " save and restore$"
simd_msg     db      13,10,"The processor supports the Streaming"
              db      " SIMD extensions to the Intel Architecture$"

not_intel    db      "t least an 80486 processor."
              db      13,10,"It does not contain a Genuine"
              db      "Intel part and as a result,"
              db      "the",13,10,"CPUID"
              db      " detection information cannot be"
              db      " determined at this time.$"
```

```
ASC_MSG      MACRO  msg
              LOCAL  ascii_done          ; local label
              add     al,30h
              cmp     al,39h              ; is it 0-9?
              jle     ascii_done
              add     al,07h
ascii_done:
              mov     byte ptr msg[20], al
              mov     dx, offset msg
              mov     ah, 9h
int          21h
ENDM
```

```
.code
.8086
```

```
print      proc

;          This procedure prints the appropriate cpuid string and
;          numeric processor presence status.  If the CPUID instruction
;          was used, this procedure prints out the CPUID info.
;          All registers are used by this procedure, none are
;          preserved.
```



```

    mov     dx, offset id_msg           ; print initial message
    mov     ah, 9h
    int     21h

    cmp     _cpuid_flag, 1             ; if set to 1, processor
    je      print_cpuid_data           ; supports CPUID instruction
                                           ; print detailed CPUID info

print_86:
    cmp     _cpu_type, 0
    jne     print_286
    mov     dx, offset cp_8086
    mov     ah, 9h
    int     21h
    cmp     _fpu_type, 0
    je      end_print
    mov     dx, offset fp_8087
    mov     ah, 9h
    int     21h
    jmp     end_print

print_286:
    cmp     _cpu_type, 2
    jne     print_386
    mov     dx, offset cp_286
    mov     ah, 9h
    int     21h
    cmp     _fpu_type, 0
    je      end_print

print_287:
    mov     dx, offset fp_287
    mov     ah, 9h
    int     21h
    jmp     end_print

print_386:
    cmp     _cpu_type, 3
    jne     print_486
    mov     dx, offset cp_386
    mov     ah, 9h
    int     21h
    cmp     _fpu_type, 0
    je      end_print
    cmp     _fpu_type, 2
    je      print_287
    mov     dx, offset fp_387
    mov     ah, 9h
    int     21h
    jmp     end_print

print_486:
    cmp     _cpu_type, 4
    jne     print_unknown             ; Intel processors will have
    mov     dx, offset cp_486sx        ; CPUID instruction
    cmp     _fpu_type, 0
```



```

        je      print_486sx
        mov     dx, offset cp_486

print_486sx:
        mov     ah, 9h
        int     21h
        jmp     end_print

print_unknown:
        mov     dx, offset cp_error
        jmp     print_486sx

print_cpuid_data:
.486
        cmp     _intel_CPU, 1                ; check for genuine Intel
        jne     not_GenuineIntel            ; processor

print_486_type:
        cmp     _cpu_type, 4                ; if 4, print 80486 processor
        jne     print_pentium_type
        mov     ax, word ptr _cpu_signature
        shr     ax, 4
        and     eax, 0fh                    ; isolate model
        mov     dx, intel_486_0[eax*2]
        jmp     print_common

print_pentium_type:
        cmp     _cpu_type, 5                ; if 5, print Pentium processor
        jne     print_pentiumpro_type
        mov     dx, offset pentium_msg
        jmp     print_common

print_pentiumpro_type:
        cmp     _cpu_type, 6                ; if 6 & model 1, print Pentium
                                           ; Pro processor
        jne     print_unknown_type
        mov     ax, word ptr _cpu_signature
        shr     ax, 4
        and     eax, 0fh                    ; isolate model
        cmp     eax, 3
        jge     print_pentiumiimodel3_type
        cmp     eax, 1
        jne     print_unknown_type          ; incorrect model number = 2
        mov     _sep_flag, 0                ; does not support Fast System
                                           ; Call

        mov     dx, offset pentiumpro_msg
        jmp     print_common

print_pentiumiimodel3_type:
        cmp     eax, 3                      ; if 6 & model 3, print Pentium
                                           ; II processor, model 3
        jne     print_pentiumiimodel5_type
        mov     ax, word ptr _cpu_signature
        and     al, 0fh                      ; isolate stepping
        cmp     al, 3
        jl      no_sep

```

```

        mov     _sep_flag, 1
        mov     dx, offset pentiumiimodel3_msg
        jmp     print_common

no_sep:
        mov     _sep_flag, 0
                                ; stepping does not support
                                ; Fast System Call

        mov     dx, offset pentiumiimodel3_msg
        jmp     print_common

print_pentiumiimodel5_type:
        cmp     eax, 5
                                ; if 6 & model 5, either Pentium
                                ; II processor, model 5, Pentium II
                                ; Xeon processor or Intel Celeron
                                ; processor, model 5

        je      celeron_xeon_detect

        cmp     eax, 7
                                ; If model 7 check cache descriptors
                                ; to determine Pentium III or Pentium III Xeon

        jne     print_celeronmodel6_type
celeron_xeon_detect:
        mov     _sep_flag, 1
                                ; Pentium II processor, model 5, Pentium II
                                ; Xeon processor, Pentium III processor,
                                ; Pentium III Xeon processor and
                                ; Intel Celeron processor support sep flag

; Is it Pentium II processor, model 5, Pentium II Xeon processor, Intel Celeron processor,
; Pentium III processor or Pentium III Xeon processor.

        OP_O
        mov     ax, word ptr _cache_eax
        OP_O
        rol     ax, 8
        mov     cx, 3

celeron_detect_eax:
        cmp     al, 40h
                                ; Is it no L2
        je      print_celeron_type
        cmp     al, 44h
                                ; Is L2 >= 1M
        jae     print_pentiumiixeon_type

        OP_O
        rol     ax, 8
        loop    celeron_detect_eax

        OP_O
        mov     ax, word ptr _cache_ebx
        mov     cx, 4

celeron_detect_ebx:
        cmp     al, 40h
                                ; Is it no L2
        je      print_celeron_type
        cmp     al, 44h
                                ; Is L2 >= 1M
        jae     print_pentiumiixeon_type

```

```

OP_O
rol    ax, 8
loop   celeron_detect_ebx

OP_O
mov     ax, word ptr _cache_ecx
mov     cx, 4

celeron_detect_ecx:
    cmp     al, 40h                ; Is it no L2
    je      print_celeron_type
    cmp     al, 44h                ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

OP_O
rol     ax, 8
loop    celeron_detect_ecx

OP_O
mov     ax, word ptr _cache_edx
mov     cx, 4

celeron_detect_edx:
    cmp     al, 40h                ; Is it no L2
    je      print_celeron_type
    cmp     al, 44h                ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

OP_O
rol     ax, 8
loop    celeron_detect_edx

    mov     dx, offset pentiumiixeon_m5_msg
    mov     ax, word ptr _cpu_signature
    shr     ax, 4
    and     eax, 0fh                ; isolate model
    cmp     eax, 5
    je      print_common
    mov     dx, offset pentiumiii_msg
    jmp     print_common

print_celeron_type:
    mov     dx, offset celeron_msg
    jmp     print_common

print_pentiumiixeon_type:
    mov     dx, offset pentiumiixeon_msg
    mov     ax, word ptr _cpu_signature
    shr     ax, 4
    and     eax, 0fh                ; isolate model
    cmp     eax, 5
    je      print_common
    mov     dx, offset pentiumiixeon_msg
    jmp     print_common

print_celeronmodel6_type:

```



```

    cmp     eax, 6                                ; if 6 & model 6, print Intel Celeron
                                                ; processor, model 6

    jne     print_pentiumiii_model8_type
    mov     _sep_flag, 1
    mov     dx, offset celeron_model6_msg
    jmp     print_common

print_pentiumiii_model8_type:
    cmp     eax, 8                                ; Pentium III processor, model 8, or
                                                ; Pentium III Xeon processor, model 8

    jb      print_unknown_type

    mov     _sep_flag, 1                          ; Each of the model 8 processors support SEP

    mov     eax, dword ptr _features_ebx
    cmp     al, 0                                ; Is brand_id supported?
    je      print_unknown_type

    mov     di, offset brand_table
    mov     cx, brand_table_size                 ; Setup pointer to brand_id table
                                                ; Get maximum entry count

next_brand:
    cmp     al, byte ptr [di]                     ; Is this the brand reported by the processor
    je      brand_found

    add     di, sizeof brand_entry
    loop    next_brand
    jmp     print_unknown_type

brand_found:
    mov     dx, word ptr [di+1]                  ; Load DX with the offset of the brand string
    jmp     print_common

print_unknown_type:
    mov     dx, offset unknown_msg               ; if n either, print unknown

print_common:
    mov     ah, 9h
    int     21h

; print family, model, and stepping
print_family:
    mov     al, _cpu_type
    ASC_MSG family_msg                          ; print family msg

print_model:
    mov     ax, word ptr _cpu_signature
    shr     ax, 4
    and     al, 0fh
    ASC_MSG model_msg                          ; print model msg

print_stepping:
    mov     ax, word ptr _cpu_signature
    and     al, 0fh
    ASC_MSG stepping_msg                      ; print stepping msg

print_upgrade:
38
```



```

    mov     ax, word ptr _cpu_signature
    test    ax, 1000h                                ; check for turbo upgrade
    jz      check_dp
    mov     dx, offset turbo_msg
    mov     ah, 9h
    int     21h
    jmp     print_features

check_dp:
    test    ax, 2000h                                ; check for dual processor
    jz      print_features
    mov     dx, offset dp_msg
    mov     ah, 9h
    int     21h

print_features:
    mov     ax, word ptr _features_edx
    and     ax, FPU_FLAG                              ; check for FPU
    jz      check_VME
    mov     dx, offset fpu_msg
    mov     ah, 9h
    int     21h

check_VME:
    mov     ax, word ptr _features_edx
    and     ax, VME_FLAG                              ; check for VME
    jz      check_DE
    mov     dx, offset vme_msg
    mov     ah, 9h
    int     21h

check_DE:
    mov     ax, word ptr _features_edx
    and     ax, DE_FLAG                              ; check for DE
    jz      check_PSE
    mov     dx, offset de_msg
    mov     ah, 9h
    int     21h

check_PSE:
    mov     ax, word ptr _features_edx
    and     ax, PSE_FLAG                              ; check for PSE
    jz      check_TSC
    mov     dx, offset pse_msg
    mov     ah, 9h
    int     21h

check_TSC:
    mov     ax, word ptr _features_edx
    and     ax, TSC_FLAG                              ; check for TSC
    jz      check_MSR
    mov     dx, offset tsc_msg
    mov     ah, 9h
    int     21h

check_MSR:

```



```

    mov    ax, word ptr _features_edx
    and    ax, MSR_FLAG                ; check for MSR
    jz     check_PAE
    mov    dx, offset msr_msg
    mov    ah, 9h
    int    21h

check_PAE:
    mov    ax, word ptr _features_edx
    and    ax, PAE_FLAG                ; check for PAE
    jz     check_MCE
    mov    dx, offset pae_msg
    mov    ah, 9h
    int    21h

check_MCE:
    mov    ax, word ptr _features_edx
    and    ax, MCE_FLAG                ; check for MCE
    jz     check_CX8
    mov    dx, offset mce_msg
    mov    ah, 9h
    int    21h

check_CX8:
    mov    ax, word ptr _features_edx
    and    ax, CX8_FLAG                ; check for CMPXCHG8B
    jz     check_APIC
    mov    dx, offset cx8_msg
    mov    ah, 9h
    int    21h

check_APIC:
    mov    ax, word ptr _features_edx
    and    ax, APIC_FLAG                ; check for APIC
    jz     check_SEP
    mov    dx, offset apic_msg
    mov    ah, 9h
    int    21h

check_SEP:
    cmp    _sep_flag, 1
    jne    print_no_sep
    mov    dx, offset sep_msg
    mov    ah, 9h
    int    21h
    jmp    check_MTRR

print_no_sep:
    mov    dx, offset no_sep_msg
    mov    ah, 9h
    int    21h

check_MTRR:
    mov    ax, word ptr _features_edx
    and    ax, MTRR_FLAG                ; check for MTRR
    jz     check_PGE
```




```

mov     dx, offset mtrr_msg
mov     ah, 9h
int     21h

check_PGE:
mov     ax, word ptr _features_edx
and     ax, PGE_FLAG                ; check for PGE
jz      check_MCA
mov     dx, offset pge_msg
mov     ah, 9h
int     21h

check_MCA:
mov     ax, word ptr _features_edx
and     ax, MCA_FLAG                ; check for MCA
jz      check_CMOV
mov     dx, offset mca_msg
mov     ah, 9h
int     21h

check_CMOV:
mov     ax, word ptr _features_edx
and     ax, CMOV_FLAG              ; check for CMOV
jz      check_PAT
mov     dx, offset cmov_msg
mov     ah, 9h
int     21h

check_PAT:
mov     eax, dword ptr _features_edx
and     eax, PAT_FLAG
jz      check_PSE36
mov     dx, offset pat_msg
mov     ah, 9h
int     21h

check_PSE36:
mov     eax, dword ptr _features_edx
and     eax, PSE36_FLAG
jz      check_PSNUM
mov     dx, offset pse36_msg
mov     ah, 9h
int     21h

check_PSNUM:
mov     eax, dword ptr _features_edx
and     eax, PSNUM_FLAG            ; check for processor serial number
jz      check_MMX
mov     dx, offset psnum_msg
mov     ah, 9h
int     21h

check_MMX:
mov     eax, dword ptr _features_edx
and     eax, MMX_FLAG              ; check for MMX technology
jz      check_FXSR

```



```

    mov     dx, offset mmx_msg
    mov     ah, 9h
    int     21h

check_FXSR:
    mov     eax, dword ptr _features_edx
    and     eax, FXSR_FLAG                ; check for FXSR
    jz      check_SIMD
    mov     dx, offset fxsr_msg
    mov     ah, 9h
    int     21h

check_SIMD:
    mov     eax, dword ptr _features_edx
    and     eax, SIMD_FLAG                ; check for Streaming SIMD
    jz      end_print                     ; Extensions
    mov     dx, offset simd_msg
    mov     ah, 9h
    int     21h

    jmp     end_print

not_GenuineIntel:
    mov     dx, offset not_intel
    mov     ah, 9h
    int     21h

end_print:
    mov     dx, offset cr_lf
    mov     ah, 9h
    int     21h
    ret
print     endp

end start
```



Example 3. Processor Identification Procedure in the C Language

```

/* Filename:  cpuid3.c                                     */
/* Copyright 1994, 1995, 1996, 1997, 1998 by Intel Corp. */
/*                                                     */
/* This program has been developed by Intel Corporation. Intel has */
/* various intellectual property rights which it may assert under */
/* certain circumstances, such as if another manufacturer's */
/* processor mis-identifies itself as being "GenuineIntel" when */
/* the CPUID instruction is executed.                      */
/*                                                     */
/* Intel specifically disclaims all warranties, express or implied, */
/* and all liability, including consequential and other indirect */
/* damages, for the use of this program, including liability for */
/* infringement of any proprietary rights, and including the */
/* warranties of merchantability and fitness for a particular */
/* purpose. Intel does not assume any responsibility for any */
/* errors which may appear in this program nor any responsibility */
/* to update it.                                           */
/*                                                     */
/* This program contains three parts:                     */
/* Part 1: Identifies CPU type in the variable _cpu_type: */
/* Part 2: Identifies FPU type in the variable _fpu_type: */
/* Part 3: Prints out the appropriate message.            */
/* This program has been tested with the Microsoft Developer Studio. */
/* If this code is compiled with no options specified and linked */
/* with the cpuid3a module, it correctly identifies the current */
/* Intel 8086/8088, 80286, 80386, 80486, Pentium(R), Pentium(R) Pro, */
/* Pentium® II, Pentium(R) II Xeon, Pentium(R) II OverDrive(R), */
/* Intel Celeron, Intel Pentium III and Intel Pentium III Xeon processors */
/* in the real-address mode.                             */
#define FPU_FLAG      0x0001
#define VME_FLAG      0x0002
#define DE_FLAG       0x0004
#define PSE_FLAG      0x0008
#define TSC_FLAG      0x0010
#define MSR_FLAG      0x0020
#define PAE_FLAG      0x0040
#define MCE_FLAG      0x0080
#define CX8_FLAG      0x0100
#define APIC_FLAG     0x0200
#define SEP_FLAG      0x0800
#define MTRR_FLAG     0x1000
#define PGE_FLAG      0x2000
#define MCA_FLAG      0x4000
#define CMOV_FLAG     0x8000
#define PAT_FLAG      0x10000
#define PSE36_FLAG    0x20000
#define PSNUM_FLAG    0x40000
#define MMX_FLAG      0x800000

```

```

#define FXSR_FLAG          0x1000000
#define SIMD_FLAG          0x2000000

extern char cpu_type;
extern char fpu_type;
extern char cpuid_flag;
extern char intel_CPU;
extern char vendor_id[12];
extern long cpu_signature;
extern long features_ecx;
extern long features_edx;
extern long features_ebx;
extern long cache_eax;
extern long cache_ebx;
extern long cache_ecx;
extern long cache_edx;

long cache_temp;
long celeron_flag;
long pentiumxeon_flag;

struct brand_entry {
    long    brand_value;
    char    *brand_string;
};

#define brand_table_size 3

struct brand_entry brand_table[brand_table_size] = {
    1, "Genuine Intel Celeron(TM) processor",
    2, "Genuine Intel Pentium(R) III processor",
    3, "Genuine Intel Pentium(R) III Xeon(TM) processor"
};

main() {
    get_cpu_type();
    get_fpu_type();
    print();
}

print() {
    int    brand_index = 0;

    printf("This system has a");
    if (cpuid_flag == 0) {
        switch (cpu_type) {
            case 0:
                printf("\n 8086/8088 processor");
                if (fpu_type) printf(" and an 8087 math coprocessor");
                break;
            case 2:
                printf("\n 80286 processor");
                if (fpu_type) printf(" and an 80287 math coprocessor");
                break;
            case 3:

```

```

        printf("n 80386 processor");
        if (fpu_type == 2)
            printf(" and an 80287 math coprocessor");
        else if (fpu_type)
            printf(" and an 80387 math coprocessor");
        break;
    case 4:
        if (fpu_type)
            printf("n 80486DX, 80486DX2 processor or 80487SX math coprocessor");
        else
            printf("n 80486SX processor");
        break;
    default:
        printf("n unknown processor");
    }
}
else {
    /* using cpuid instruction */
    if (intel_CPU) {
        if (cpu_type == 4) {
            switch ((cpu_signature >> 4) & 0xf) {
                case 0:
                case 1:
                    printf(" Genuine Intel486(TM) DX processor");
                    break;
                case 2:
                    printf(" Genuine Intel486(TM) SX processor");
                    break;
                case 3:
                    printf(" Genuine IntelDX2(TM) processor");
                    break;
                case 4:
                    printf(" Genuine Intel486(TM) processor");
                    break;
                case 5:
                    printf(" Genuine IntelSX2(TM) processor");
                    break;
                case 7:
                    printf(" Genuine Write-Back Enhanced \
                        IntelDX2(TM) processor");
                    break;
                case 8:
                    printf(" Genuine IntelDX4(TM) processor");
                    break;
                default:
                    printf(" Genuine Intel486(TM) processor");
            }
        }
        else if (cpu_type == 5)
            printf(" Genuine Intel Pentium(R) processor");
        else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 1))
            printf(" Genuine Intel Pentium(R) Pro processor");
        else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 3))
            printf(" Genuine Intel Pentium(R) II processor, model 3");
        else if (((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 5)) ||
            ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 7)))
    }
}

```

```

{
    celeron_flag = 0;
    pentiumxeon_flag = 0;
    cache_temp = cache_eax & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_eax & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_eax & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;
    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;
    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF;
    if (cache_temp == 0x40)
        celeron_flag = 1;
    if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))

```

```

        pentiumxeon_flag = 1;

cache_temp = cache_ecx & 0xFF00;
if (cache_temp == 0x4000)
    celeron_flag = 1;
if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
    pentiumxeon_flag = 1;

cache_temp = cache_ecx & 0xFF;
if (cache_temp == 0x40)
    celeron_flag = 1;
if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
    pentiumxeon_flag = 1;

cache_temp = cache_edx & 0xFF000000;
if (cache_temp == 0x40000000)
    celeron_flag = 1;
if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
    pentiumxeon_flag = 1;

cache_temp = cache_edx & 0xFF0000;
if (cache_temp == 0x400000)
    celeron_flag = 1;
if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
    pentiumxeon_flag = 1;

cache_temp = cache_edx & 0xFF00;
if (cache_temp == 0x4000)
    celeron_flag = 1;
if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
    pentiumxeon_flag = 1;

cache_temp = cache_edx & 0xFF;
if (cache_temp == 0x40)
    celeron_flag = 1;
if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
    pentiumxeon_flag = 1;

if (celeron_flag == 1)
    printf(" Genuine Intel Celeron(TM) processor, model 5");
else
{
    if (pentiumxeon_flag == 1) {
        if (((cpu_signature >> 4) & 0x0f) == 5)
            printf(" Genuine Intel Pentium(R) II Xeon(TM) processor");
        else
            printf(" Genuine Intel Pentium(R) III Xeon(TM) processor,");
        printf(" model 7");
    }
    else {
        if (((cpu_signature >> 4) & 0x0f) == 5) {
            printf(" Genuine Intel Pentium(R) II processor, model 5 ");
            printf("or Intel Pentium(R) II Xeon processor");
        }
        else {
            printf(" Genuine Intel Pentium(R) III processor, model 7");

```



```

        printf(" or Intel Pentium(R) III Xeon(TM) processor,");
        printf(" model 7");
    }
}
}
else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 6))
    printf(" Genuine Intel Celeron(TM) processor, model 6");
else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) >= 8) &&
        ((features_ebx & 0xff) != 0)) {
    while ((brand_index < brand_table_size) &&
        ((features_ebx & 0xff) != brand_table[brand_index].brand_value))
        brand_index++;
    if (brand_index < brand_table_size)
        printf("%s", brand_table[brand_index].brand_string);
    else
        printf("n unknown Genuine Intel processor");
}
else
    printf("n unknown Genuine Intel processor");
printf("\nProcessor Family: %X", cpu_type);
printf("\nModel:      %X", (cpu_signature >> 4) & 0xf);
printf("\nStepping:    %X\n", cpu_signature & 0xf);
if (cpu_signature & 0x1000)
    printf("\nThe processor is an OverDrive(R) processor");
else if (cpu_signature & 0x2000)
    printf("\nThe processor is the upgrade processor in a dual processor system");
if (features_edx & FPU_FLAG)
    printf("\nThe processor contains an on-chip FPU");
if (features_edx & VME_FLAG)
    printf("\nThe processor supports Virtual Mode Extensions");
if (features_edx & DE_FLAG)
    printf("\nThe processor supports the Debugging Extensions");
if (features_edx & PSE_FLAG)
    printf("\nThe processor supports Page Size Extensions");
if (features_edx & TSC_FLAG)
    printf("\nThe processor supports Time Stamp Counter");
if (features_edx & MSR_FLAG)
    printf("\nThe processor supports Model Specific Registers");
if (features_edx & PAE_FLAG)
    printf("\nThe processor supports Physical Address Extension");
if (features_edx & MCE_FLAG)
    printf("\nThe processor supports Machine Check Exceptions");
if (features_edx & CX8_FLAG)
    printf("\nThe processor supports the CMPXCHG8B instruction");
if (features_edx & APIC_FLAG)
    printf("\nThe processor contains an on-chip APIC");
if (features_edx & SEP_FLAG) {
    if ((cpu_type == 6) && ((cpu_signature & 0xff) < 0x33))
        printf("\nThe processor does not support the Fast System Call");
    else
        printf("\nThe processor supports the Fast System Call");
}
if (features_edx & MTRR_FLAG)
    printf("\nThe processor supports the Memory Type Range Registers");
if (features_edx & PGE_FLAG)

```



```

        printf("\nThe processor supports Page Global Enable");
    if (features_edx & MCA_FLAG)
        printf("\nThe processor supports the Machine Check Architecture");
    if (features_edx & CMOV_FLAG)
        printf("\nThe processor supports the Conditional Move Instruction");
    if (features_edx & PAT_FLAG)
        printf("\nThe processor supports the Page Attribute Table");
    if (features_edx & PSE36_FLAG)
        printf("\nThe processor supports 36-bit Page Size Extension");
    if (features_edx & PSNUM_FLAG)
        printf("\nThe processor supports the processor serial number");
    if (features_edx & MMX_FLAG)
        printf("\nThe processor supports Intel Architecture MMX(TM) technology");
    if (features_edx & FXSR_FLAG)
        printf("\nThe processor supports the Fast floating point save and restore");
    if (features_edx & SIMD_FLAG)
        printf("\nThe processor supports the Streaming SIMD extensions to the Intel Architecture");
    }
    else {
        printf("t least an 80486 processor. ");
        printf("\nIt does not contain a Genuine Intel part and as a result, the ");
        printf("\nCUID detection information cannot be determined at this time.");
    }
    }
    printf("\n");
}

```